A GPU Algorithm for Greedy Graph Matching

B. O. Fagginger Auer and R. H. Bisseling

Mathematics Institute, Utrecht University,
Budapestlaan 6, 3584 CD, Utrecht, the Netherlands
B.O.FaggingerAuer@uu.nl
R.H.Bisseling@uu.nl

Abstract. Greedy graph matching provides us with a fast way to coarsen a graph during graph partitioning. Direct algorithms on the CPU which perform such greedy matchings are simple and fast, but offer few handholds for parallelisation. To remedy this, we introduce a fine-grained shared-memory parallel algorithm for maximal greedy matching, together with an implementation on the GPU, which is faster (speedups up to 6.8 for random matching and 5.6 for weighted matching) than the serial CPU algorithms and produces matchings of similar (random matching) or better (weighted matching) quality.

1 Introduction

We propose a fine-grained shared-memory parallel algorithm for generating greedy matchings of undirected graphs. This algorithm was inspired by the parallel graph coarsening algorithm discussed in [8, Sec. 3.2] and follows a paradigm similar to that of the auction algorithm [4] for bipartite graphs (implemented on the GPU in [17]), but applying to general greedy matchings in arbitrary undirected graphs. The GPU has been used to accelerate solving of sparse problems, e.g. in the CUSP library [3], which provides sparse linear algebra and graph computations using CUDA.

Graph matchings have applications in minimising power consumption in dynamic wireless networks [19], heuristics for solving the travelling salesman problem [9], and organ donation [16]. Our primary interest however, will be the coarsening of graphs and hypergraphs (where edges may contain more than two vertices). During coarsening, we first match neighbouring vertices in a (hyper)graph and then merge matched pairs of vertices into a single vertex, which yields a coarser version of the (hyper)graph. Repeatedly coarsening a (hyper)graph gives a multi-level hierarchy of increasingly coarser approximations of the original (hyper)graph, which is useful for (hyper)graph partitioning, e.g. in the context of sparse matrix–vector multiplication [5,18] or LU decomposition of sparse matrices [1,7]. The following definitions have been set up such that they can easily be generalised to hypergraphs.

Graphs will be denoted by $G = (V, E)$, where $V \subseteq \mathbf{N}$ is the set of *vertices* and E the set of *edges* of the graph (all $e \in E$ are of the form $e = \{v, w\}$ for

some $v, w \in V$). For $v \in V$, we denote the *collection of neighbours of* v by

$$V_v := \{w \in V \mid \exists e \in E : v, w \in e\} \setminus \{v\}.$$

The graph G is *weighted* if it is provided with a function $\omega : E \to \mathbf{R}_{>0}$ assigning a weight $\omega(e) > 0$ to each edge $e \in E$.

A *matching of* G is a map $\pi : V \to \mathbf{N}$ such that

1. for all $v \in V$ there exists at most one $w \in V \setminus \{v\}$ such that $\pi(v) = \pi(w)$ (we match at most two vertices to each other),
2. for all $v, w \in V$, $v \neq w$, if $\pi(v) = \pi(w)$, then $v \in V_w$ and $w \in V_v$ (we only match neighbouring vertices).

We consider two different vertices $v, w \in V$ to be *matched to each other* if $\pi(v) = \pi(w)$. If we cannot match any more vertices without breaking one of these two conditions, we call π *maximal*. If G is weighted, then the *weight* ω_π *of* π is defined as the sum of the weights of all edges included in the matching:

$$M_\pi := \{\{v, w\} \in E \mid \pi(v) = \pi(w), v \neq w\}, \qquad \omega_\pi := \sum_{e \in M_\pi} \omega(e).$$

2 Serial matching

We will consider simple greedy random matching, as outlined in Alg. 1. For this algorithm we use $\pi(v) = \infty$ to indicate that the vertex v is unmatched.

Algorithm 1 Serially creates a matching of a graph $G = (V, E)$ with $V \subseteq \mathbf{N}$ by constructing $\pi : V \to \mathbf{N} \cup \{\infty\}$.

```
1: Randomise the order of the vertices in V.
2: for v ∈ V do
3:     π(v) ← ∞;
4: for v ∈ V do
5:     if π(v) = ∞ then
6:         w ← select(v, V_v ∩ π^{-1}({∞}));
7:         if w ≠ ∞ then
8:             π(v) ← min{v, w};
9:             π(w) ← min{v, w};
```

The function $\mathbf{select}(v, W)$ is defined for vertices $v \in V$ and collections of neighbours $W \subseteq V_v$. Should W be empty, then $\mathbf{select}(v, W) = \infty$, otherwise $\mathbf{select}(v, W) = w$ for some neighbour $w \in W$ of v. Choosing different prescriptions for selecting neighbours gives us different kinds of matchings. Here, we consider two options for **select**: *random matching* and *weighted matching*.

For random matching, $\mathbf{select}(v, W)$ returns the first available $w \in W$. Because we randomise vertex order, this amounts to matching vertices to random neighbours, while providing an early exit for the selection mechanism.

For weighted matching, **select**(v, W) returns a neighbour $w \in W$ with $\omega(\{v, w\}) = \max_{u \in W} \omega(\{v, u\})$. Here, the selection process takes longer: every neighbour needs to be considered to find the heaviest edge originating from v.

Note that in either case Alg. 1 produces maximal matchings. This ensures that the number of matched vertices is at least half of the maximum possible number of matched vertices when considering all possible matchings.

Other greedy matching strategies such as dynamic minimum degree (vertices with fewest unmatched neighbours are matched first) or Karp–Sipser [10] (vertices with a single unmatched neighbour are matched first) are not considered, because dynamically keeping track of all vertex degrees leads to serialisation. A more in-depth discussion and comparison of such matching strategies can be found in [11]. A distributed-memory parallel implementation of the Karp–Sipser algorithm is presented in [13].

2.1 Matching by decreasing edge weights

We will also compare weighted matchings generated by our GPU algorithm with weighted matchings generated by Alg. 2.

Algorithm 2 Serially creates a weighted matching of a weighted graph $G = (V, E)$ with $V \subseteq \mathbf{N}$ and weights $\omega : E \to \mathbf{R}_{>0}$ by constructing $\pi : V \to \mathbf{N} \cup \{\infty\}$.

```
1: for v ∈ V do
2:     π(v) ← ∞;
3: for {v, w} ∈ E in order of decreasing ω({v, w}) do
4:     if π(v) = ∞, π(w) = ∞, and v ≠ w then
5:         π(v) ← min{v, w};
6:         π(w) ← min{v, w};
```

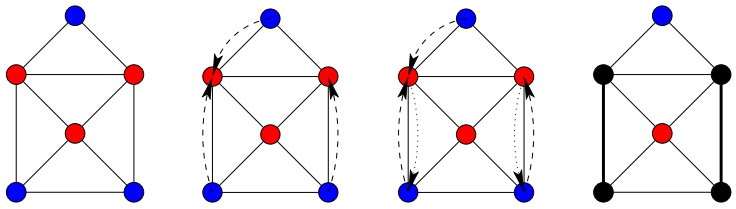
Alg. 2 ensures that we always match the vertices belonging to an edge with maximum weight in the entire graph, in contrast to weighted matching by Alg. 1 where the edge with maximum weight originating from a random vertex is matched. Because of this, Alg. 2 is $\frac{1}{2}$-optimal, i.e. the weight ω_π of the matching π generated by Alg. 2 is guaranteed to be at least half of the maximum weight that any matching of this graph can attain. A distributed-memory parallel algorithm for weighted matching is given in [12]; this algorithm is based on locally dominant edges and is also $\frac{1}{2}$-optimal.

3 Parallel matching

A problem with Alg. 1 is its serial nature: in order to prevent matching more than two vertices to each other, we seemingly have to consider vertices one-by-one. To be able to match vertices simultaneously, while still satisfying the matching criteria, we propose Alg. 3, which permits us to evaluate **select** in parallel for many vertices.

Algorithm 3 Creates a matching of a graph $G = (V, E)$, with $V \subseteq \mathbf{N}$, in parallel by constructing $\pi : V \to \mathbf{N} \cup \{\textbf{blue}, \textbf{red}, \textbf{dead}\}$.

```
 1: for all v ∈ V parallel do
 2:     π(v) ← blue;
 3: done ← false;
 4: while not done do
 5:     {Assign vertex colours:}
 6:     done ← true;
 7:     for all v ∈ V parallel do
 8:         if π(v) ∈ {blue, red} then
 9:             done ← false;
10:             π(v) ← colour(v);
11:     {Blue vertices propose to red vertices:}
12:     for all v ∈ V parallel do
13:         if π(v) = blue then
14:             if V_v ∩ π^{-1}({blue, red}) = ∅ then
15:                 σ(v) ← dead;
16:             else
17:                 σ(v) ← select(v, V_v ∩ π^{-1}({red}));
18:         else
19:             σ(v) ← ∞;
20:     {Red vertices respond to blue vertices:}
21:     for all v ∈ V parallel do
22:         if π(v) = red then
23:             if V_v ∩ π^{-1}({blue, red}) = ∅ then
24:                 σ(v) ← dead;
25:             else
26:                 σ(v) ← select(v, V_v ∩ π^{-1}({blue}) ∩ σ^{-1}({v}));
27:     {Match mutual proposals:}
28:     for all v ∈ V parallel do
29:         if σ(v) = dead then
30:             π(v) ← dead;
31:         else if σ(v) ≠ ∞ then
32:             if σ(σ(v)) = v then
33:                 π(v) ← min{v, σ(v)};
```



(a) Colour. (b) Propose. (c) Respond. (d) Match.

Fig. 1. Illustration of one iteration of Alg. 3's main loop: (a) we colour all vertices blue or red; (b) let the blue vertices propose to the red vertices; (c) let the red vertices respond to one of these proposals; (d) and match the mutual proposals.

For this algorithm $\pi(v) \in \{\textbf{blue}, \textbf{red}, \textbf{dead}\}$ indicates that v has not been matched. The function **colour**(v) determines for vertices $v \in V$ whether they are put into the **blue** or the **red** group. The **for all ... parallel do** construct indicates a for-loop where each iteration can be executed completely independently. These for-loops make Alg. 3 suitable for a GPU implementation, where each independent loop iteration (corresponding to a vertex) is mapped to a different GPU thread. Furthermore, π and σ can be kept on the GPU during the iterations of Alg. 3, such that communication between the CPU and GPU is limited to only the start and the end of the matching process.

Alg. 3 starts by marking all vertices $v \in V$ as **blue** (line 2), such that they are unmatched. Then, we enter the main loop (line 4) and colour each unmatched vertex **blue** or **red** (line 10, this is irrespective of the current colour of the vertex). All **blue** vertices propose to **red** neighbours, chosen by **select** (line 17). Vertices without unmatched neighbours are flagged as being **dead**. **Red** vertices then consider proposals made to them by their neighbours, and respond to one of them, chosen by **select** (line 26). Here, data thrashing due to parallel reads and writes to σ is avoided by only checking whether $\sigma(w) = v$ for neighbours $w \in V_v$ with $\pi(w) = \textbf{blue}$. After this, we match all vertices that have compatible proposals and responses (line 28). Vertices that were flagged as dead receive a special matching value (**dead**) so that they are no longer considered for matching in subsequent iterations. We restart the main loop and reassign unmatched vertices to either the **blue** or **red** group. The main loop is repeated until we obtain a maximal matching.

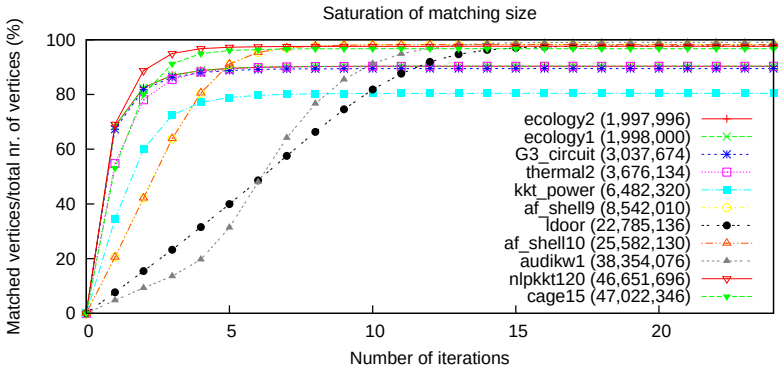


Fig. 2. The ratio between the number of matched vertices and the total number of vertices, as a function of the number of iterations of the while-loop at line 4 of Alg. 3. The number after each graph name indicates the number of edges.

The effect of iterating the main loop of Alg. 3 can be seen in Fig. 2. Here we observe that the number of unmatched vertices decreases rapidly as the number of iterations increases, stabilising when the matching is maximal. Note that the matching is maximal when all vertices are either matched or **dead**.

Therefore, we keep track of a 'done' flag in Alg. 3, which becomes **true** when $\pi^{-1}(\{\textbf{blue}, \textbf{red}\}) = \emptyset$. Because we only need to store a fixed value in 'done' at line 9, we can do this directly in parallel without having to resort to atomic operations (an atomic compare-and-swap halved performance during experiments).

3.1 Vertex labelling

It is important that the function **colour** finds different **blue** and **red** groups every iteration, because otherwise we can get stuck in situations where a non-maximal matching is not enlarged. A direct way to define this function is to determine the **blue** and **red** groups by randomly assigning each vertex to the **blue** group with probability p and to the **red** group with probability $1 - p$, i.e.

$$\textbf{colour}(v) = \begin{cases} \textbf{blue} \text{ with probability } p \in [0,1], \\ \textbf{red} \quad \text{otherwise.} \end{cases} \tag{1}$$

Intuitively, we should ensure that the **blue** and **red** groups are approximately of equal size (by picking $p = \frac{1}{2}$, similar to [8, Sec. 3.2]) so that all unmatched vertices have a good chance of possessing a neighbour of a different colour and are therefore able to propose or respond in the current iteration. This leads to a large number of matched or **dead** vertices, which will speed up later iterations.

Let us make this more precise by considering random matching in a random graph G with vertices $V = \{1, \ldots, n\}$, where an edge between two vertices $v, w \in V$ exists with probability $P(\{v, w\} \in E) = d$ for a fixed density parameter $d \in [0, 1]$. During a single iteration of Alg. 3 we match a number of vertices equal to twice the number N of **red** vertices that receive a proposal from a **blue** neighbour, i.e.

$$\begin{aligned} N &= \sum_{v \in V} P(\pi(v) = \textbf{red})\, P(v \text{ is proposed to } \mid \pi(v) = \textbf{red}) \\ &= \sum_{v \in V} P(\pi(v) = \textbf{red}) \left(1 - \prod_{w \in V \setminus \{v\}} (1 - P(w \text{ proposes to } v \mid \pi(v) = \textbf{red}))\right) \\ &= \sum_{v \in V} P(\pi(v) = \textbf{red}) \left(1 - \prod_{w \in V \setminus \{v\}} \left(1 - \frac{P(\pi(w) = \textbf{blue})\, P(\{v, w\} \in E)}{\text{nr. of } \textbf{red} \text{ neighb. of } w}\right)\right). \end{aligned}$$

We now approximate the number of **red** neighbours of w by its average $1 + (1 - p)\,(d\,(n-1) - 1)$ (since v is already a **red** neighbour of w). This gives

$$N \approx N^* := n\,(1-p) \left(1 - \left(1 - \frac{p\,d}{1 + (1-p)\,(d\,(n-1) - 1)}\right)^{n-1}\right).$$

The approximate expected fraction of matched vertices in a large random graph for a single iteration of Alg. 3 then equals

$$\lim_{n \to \infty} \frac{2\,N^*}{n} = 2\,(1-p) \left(1 - e^{-\frac{p}{1-p}}\right). \tag{2}$$

This function is maximal for $p \in [0,1]$ satisfying $1-p = e^{-\frac{p}{1-p}}\,(2-p)$, yielding $p \approx 0.53406$, independent of the density d. Therefore, this choice of p yields the largest number of matched vertices per iteration, and hence the shortest running time of Alg. 3, regardless of the edge density of the random graph. Because of this, we expect such a p also to work well for non-random graphs, which we confirmed experimentally for `ecology1` in Fig. 3.

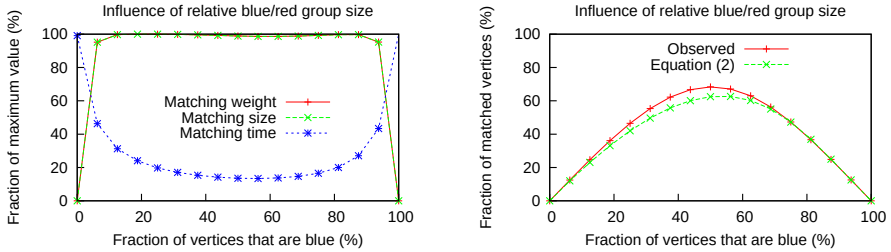


Fig. 3. The effect of the probability p from eq. (1) (left) on the absolute matching size, weight, and time required to generate the matching, rescaled to a range of 100%, and (right) on the observed and theoretical, eq. (2), fraction of matched vertices during the first iteration of Alg. 3 for the matrix `ecology1`.

3.2 Random vertex assignment

To evaluate eq. (1) in parallel on the GPU we use the MD5 message digest algorithm [15]. This algorithm calculates a 128-bit value, the *MD5 hash*, of a given sequence of bits, such that small changes in this sequence in general result in a completely different MD5 hash. A sequence of bits is converted to an MD5 hash by padding the sequence such that its number of bits is a multiple of 512, and then adding the contribution of each 512-bit chunk of the padded sequence to the hash.

To employ the MD5 algorithm as a random number generator we generate a single random number r on the CPU, which we pass to the GPU as a parameter for all threads (we again create a GPU thread for each vertex $v \in V$). Then, we create for each 32-bit vertex number v a 512-bit array consisting of $\{v, r\,v, \ldots, r^{15}\,v\}$ and calculate the hash of this array, which we normalise to obtain a pseudorandom number in $[0,1]$ for eq. (1). Changing either v or r will result in a different array and therefore a completely different hash. By generating different values r, we can therefore create completely different assignments of the unmatched vertices of the graph, for each iteration of Alg. 3.

To improve performance we only use the first quarter of the MD5 algorithm, which did not reduce the quality of the matchings during experiments. This makes Alg. 4 fast and parallel, requires only a small amount of thread-independent storage (we do not store $\{v, r\,v, \ldots, r^{15}\,v\}$ explicitly), and yields reproducible vertex colourings.

Algorithm 4 Implementation of $\mathbf{colour}_r(v)$ for Alg. 3, based on [15, Sec. 3.4]. Here $v \in \mathbf{N}$ is a vertex, $r \in \mathbf{N}$ is a parameter, $p \in [0, 1]$ (eq. (1)), and K and R are MD5 constants and shift amounts, kept in constant GPU memory.

Initialise hash as h_0, h_1, h_2, h_3.
Let $a_0 \leftarrow h_0$, $a_1 \leftarrow h_1$, $a_2 \leftarrow h_2$, $a_3 \leftarrow h_3$.
for $i = 0$ to 15 **do**
    $a_4 \leftarrow (a_1$ **and** $a_2)$ **or** ((**not** a_1) **and** $a_3)$;
    $a_5 \leftarrow a_3$, $a_3 \leftarrow a_2$, $a_2 \leftarrow a_1$;
    $a_1 \leftarrow a_1 + \mathbf{rol}(a_0 + a_4 + K(i) + v, R(i))$; (bitwise **ro**tate **l**eft)
    $a_0 \leftarrow a_5$;
    Add $a_0, \ldots, a_3$ to $h_0, \ldots, h_3$.
    $v \leftarrow r\, v$;
if $(h_0 + h_1 + h_2 + h_3) \mod 2^{32} < p\, 2^{32}$ **then**
    return **blue**;
else
    return **red**;

4 Results

For graph coarsening, it is important to randomise the ordering of the vertices of the graph to ensure that we do not get stuck in star graphs [14, Sec. 5.3]. Therefore, we randomly permute all vertices on the CPU after the graph has been read from disk (as was also done in the experiments in [11]), where we use the same permutation for benchmarking the serial and parallel algorithms. Randomisation of the vertices will decrease performance, because it prevents coalesced reading on the GPU when looping over vertex neighbours. As we are interested in the performance of the greedy matching process itself, permuting the graph and I/O transfer have not been included in the recorded timings. CPU to GPU transfer takes up, on average, 39% of the time.

The actual implementation of Alg. 3 was done with both NVIDIA's Compute Unified Device Architecture (CUDA) library version 3.1.2 and Intel's Threading Building Blocks (TBB) library version 3.0 in C++, compiled with g++ version 4.1.2 using O3 optimisation flags. For the CUDA implementation, static graph data (i.e. neighbour ranges, indices, and edge weights) were placed in one-dimensional textures to improve cache use. Dynamic data (i.e. π and σ) were placed in one-dimensional arrays, such that using a one-dimensional thread distribution (one thread per vertex and a CUDA block size of 256) gives us coalesced data writing everywhere in the algorithm. For more implementation details, we would like to refer the reader to the source code of the discussed algorithms, which is freely available at `http://www.staff.science.uu.nl/~faggi101/`.

For weighted matching, we use 425 symmetric matrices from the University of Florida sparse matrix collection [6], where the edge weights are set to the absolute value of the corresponding matrix entry. For random matching this set is augmented with the graphs from the 10th DIMACS challenge on graph

partitioning [2], which do not possess edge weights. This gives us a large, unbiased test set of matrices arising from real-world applications.

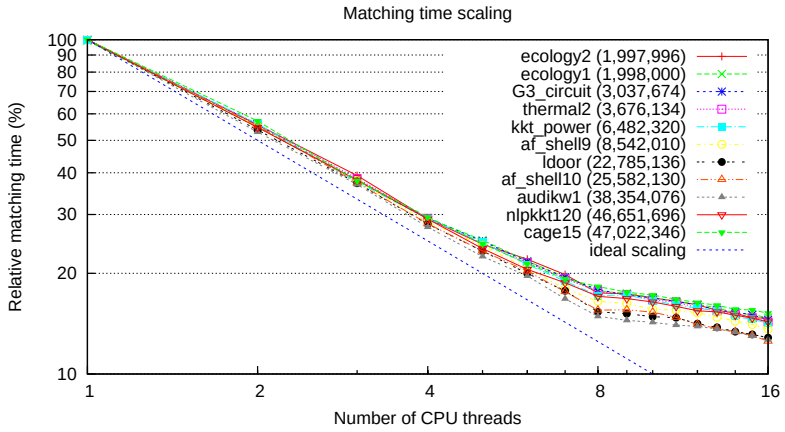


Fig. 4. Scaling of the random matching time of Alg. 3 with the number of TBB threads, on a dual quad-core CPU with hyperthreading (8 physical cores) in a log-log plot. The matching time is relative to the matching time required by Alg. 3 on a single core.

The experiments were performed on a computer equipped with two quad-core 2.4 GHz Intel Xeon E5620 processors with hyperthreading, 24 GiB RAM, and an NVIDIA Tesla C2050 with 2687 MiB global memory. We measured the scaling of the TBB implementation of Alg. 3 with respect to the number of threads used by the CPU in Fig. 4 and compared both the CUDA and TBB (using 16 threads) implementations to the serial matching algorithms (Alg. 1 and Alg. 2) in Fig. 5. Fig. 5 shows the ratios of the average (over 32 random permutations of the graph vertices) matching size, time, and weight, together with error bars of one standard deviation. From these results, we observe the following:

- Alg. 3 scales well as we increase the used number of threads. The test system possesses 8 physical cores, but up to 16 threads with hyperthreading, which explains good speedups up to 8 threads and smaller speedups thereafter.
- The quality of the generated random matchings by Alg. 3 is comparable to that of the serial algorithm: for both CUDA and TBB the average matching size ratio is more than 99%.
- Weighted matching with Alg. 3, for both CUDA and TBB, yields higher quality matchings than Alg. 1 (average matching weight ratio 115%), but lower quality matchings than Alg. 2 (ratio of 85%). This is not surprising, since Alg. 2 always picks the globally heaviest edge, whereas Alg. 1 and Alg. 3 pick heavy edges locally. Furthermore, Alg. 1 does this one-sidedly, whereas Alg. 3 performs a two-sided comparison (both proposers and responders pick the heaviest neighbour), which leads to an increase in matching weight.

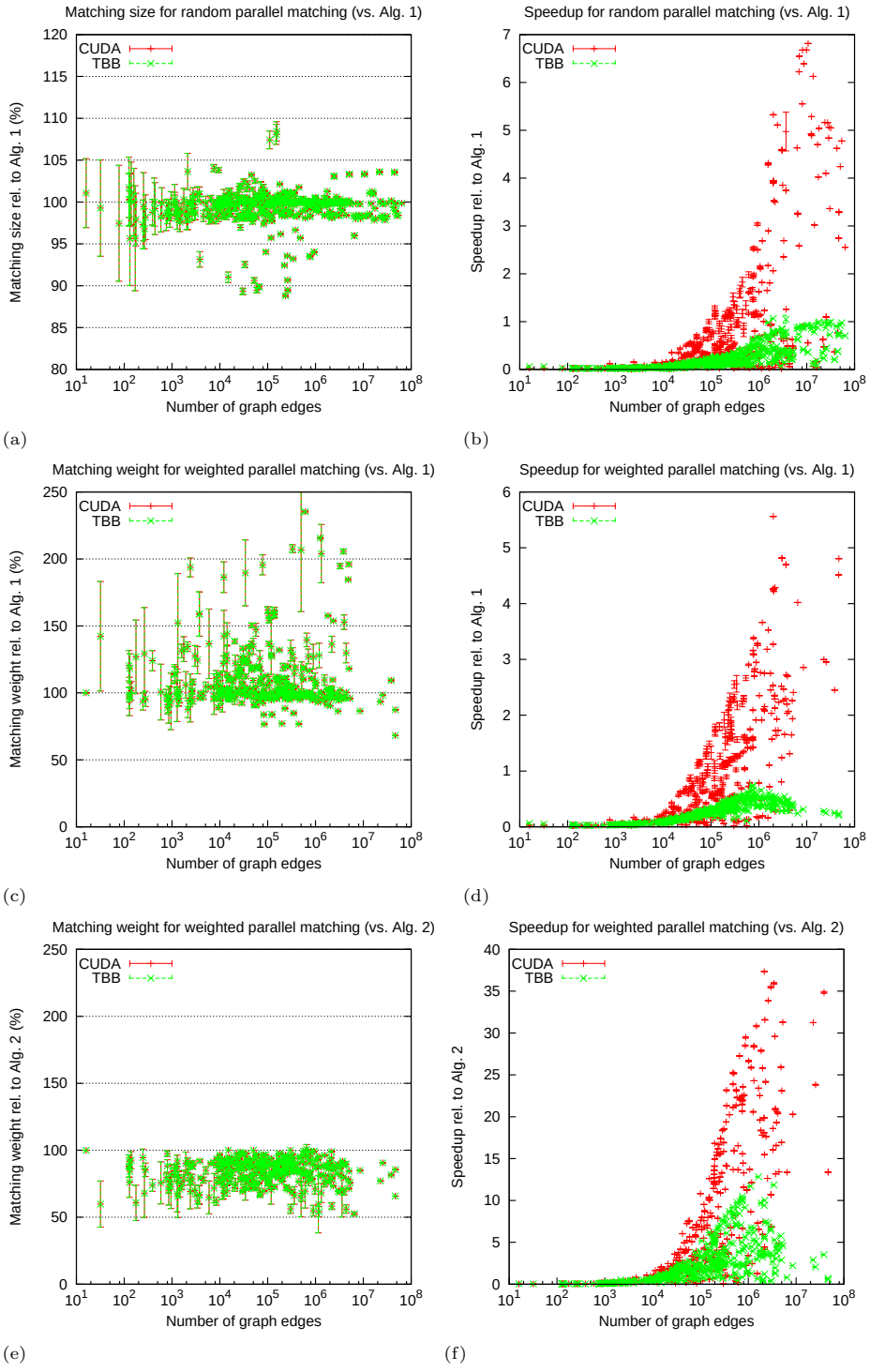


Fig. 5. Comparison between the serial matching algorithms (Alg. 1 and Alg. 2) and Alg. 3 implemented in CUDA on the GPU and in TBB on a multi-core CPU.

- In Fig. 5 we see that Alg. 3 does not obtain the same speedup for all graphs. This is related to the ratio of the maximum and minimum degree of the vertices of the graph, $(\max_{v \in V} |V_v|) \,/\, (\min_{w \in V} |V_w|)$. When this ratio is large, vertices with high degree will keep a small number of CUDA threads occupied for a long time, while the other kernels have already finished, leading to a low occupancy of the GPU and decreased performance.
- The speedups increase as the graphs become larger. For CUDA, Alg. 3 reaches speedups up to 6.8, 5.6, and 37 compared to random matching with Alg. 1 and weighted matching with Alg. 1 and Alg. 2. However, for TBB we only reach speedups up to 1.1, 0.7, and 13.

Most of the time in Alg. 3 is spent loading and storing data, instead of performing calculations. This is confirmed by the NVIDIA CUDA profiler for random matching of `ecology1`, where the instruction-to-byte ratios are equal to 2.36 and 1.31 (according to the profiler, they should be close to 4.06) for proposing and responding to proposals in Alg. 3: this makes the algorithm bandwidth limited. Non-coalesced memory access due to randomisation is reflected in a low texture-cache hit rate, which is 35% for proposing, and 3% for responding, but we do utilize 70% and 82%, respectively, of the maximum available global memory bandwidth. This explains the fact that the GPU, with its much larger bandwidth (144 GB/s for a Tesla C2050's global memory vs. 17 GB/s for DDR3 RAM), performs better than the CPU TBB implementation, and that for weighted matching (where the edge weights also need to be read) the speedups are smaller, because memory traffic is increased. We therefore expect performance to be increased further when **select** involves a more compute-intensive assessment of each of the vertex's neighbours.

5 Conclusion

We have described a fine-grained shared-memory parallel algorithm for greedy graph matching (Alg. 3) and created a GPU implementation of this algorithm to compare it with serial greedy matching on the CPU (Alg. 1 and 2). For random matching, Alg. 3 provides maximal matchings of similar quality as Alg. 1, it is slower for smaller graphs ($< 10^5$ edges), but becomes increasingly faster as the number of edges increases (up to a speedup factor of 6.8). For weighted matching of large graphs, Alg. 3 offers both better performance (speedups up to 5.6) and better quality than Alg. 1, while compared to Alg. 2 we sacrifice matching quality for a much better performance (speedups up to 37). Alg. 3 performs much better on the GPU than on the multi-core CPU because of the GPU's superior memory bandwidth. These results were obtained for a large set of graphs arising from real-world applications.

We are interested in applying this algorithm in the context of (hyper)graph coarsening [18] and anticipate that there, with more complicated ways for vertices to select desired neighbours to be matched to, the ability of Alg. 3 to perform many of these selections in parallel will lead to higher speedups.

Acknowledgements

We would like to thank Albert-Jan Yzelman and Jaap Eldering for their help in refining the algorithm, Vianney Govers for extensive technical support, and the Little Green Machine project, `http://littlegreenmachine.org/`, for enabling us to run our benchmarks on their hardware under project NWO-M 612.071.305.

References

1. Aykanat, C., Pinar, A., Çatalyürek, U.V.: Permuting sparse rectangular matrices into block-diagonal form. SIAM J. Sci. Comput. 25(6), 1860–1879 (2004)
2. Bader, D.A., Sanders, P., Wagner, D., Meyerhenke, H., Hendrickson, B., Johnson, D.S., Walshaw, C.: 10th DIMACS implementation challenge - graph partitioning and graph clustering (2011), `http://www.cc.gatech.edu/dimacs10/index.shtml`
3. Bell, N., Garland, M.: Cusp: Generic parallel algorithms for sparse matrix and graph computations (2010), `http://cusp-library.googlecode.com`, version 0.1.0
4. Bertsekas, D.P.: A distributed asynchronous relaxation algorithm for the assignment problem. In: 24th IEEE CDC. vol. 24, pp. 1703–1704 (1985)
5. Çatalyürek, U.V., Aykanat, C.: Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. IEEE Trans. Par. Dist. Syst. 10(7), 673–693 (1999)
6. Davis, T.A., Hu, Y.: The University of Florida Sparse Matrix Collection. ACM TOMS 38(1) (2011)
7. Grigori, L., Boman, E.G., Donfack, S., Davis, T.A.: Hypergraph-based unsymmetric nested dissection ordering for sparse LU factorization. SIAM J. Sci. Comput. 32(6), 3426–3446 (2010)
8. Her, J.H., Pellegrini, F.: Efficient and scalable parallel graph partitioning (2010), Parallel Computing (to appear)
9. Kahng, A.B., Reda, S.: Match twice and stitch: a new TSP tour construction heuristic. Operations Research Letters 32(6), 499–509 (2004)
10. Karp, R.M., Sipser, M.: Maximum matchings in sparse random graphs. In: Proc. 22nd FOCS. pp. 364–375 (1981)
11. Langguth, J., Manne, F., Sanders, P.: Heuristic initialization for bipartite matching problems. J. Exp. Algorithmics 15(1.3), 1.1–1.22 (2010)
12. Manne, F., Bisseling, R.H.: A parallel approximation algorithm for the weighted maximum matching problem. In: PPAM 2007. LNCS, vol. 4967, pp. 708–717 (2008)
13. Patwary, M.A., Bisseling, R.H., Manne, F.: Parallel greedy graph matching using an edge partitioning approach. In: Proc. HLPP 2010. pp. 45–54. ACM (2010)
14. Preis, R.: Analyses and design of efficient graph partitioning methods. HNI-Verlagsschriftenreihe, Heinz Nixdorf Inst, Univ. Paderborn (2001)
15. Rivest, R.L.: The MD5 message-digest algorithm (1992), Internet RFC 1321
16. Segev, D.L., Gentry, S.E., Warren, D.S., Reeb, B., Montgomery, R.A.: Kidney paired donation and optimizing the use of live donor organs. JAMA 293(15), 1883–1890 (2005)
17. Vasconcelos, C.N., Rosenhahn, B.: Bipartite graph matching computation on GPU. In: Proc. EMMCVPR 2009. pp. 42–55. Springer-Verlag, Berlin, Heidelberg (2009)
18. Vastenhouw, B., Bisseling, R.H.: A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. SIAM Rev. 47(1), 67–95 (2005)
19. Xing, G., Lu, C., Zhang, Y., Huang, Q., Pless, R.: Minimum power configuration for wireless communication in sensor networks. ACM Trans. Sen. Netw. 3(2) (2007)